Android Gradle 插件中文指南

Yourtion

Published with GitBook

目錄

Android Gradle 插件中文指南

翻译

内容由飞雪无情提供翻译

原文地址：https://github.com/rujews/android-tech-docs/tree/master/new-build-system/user-guide

英文原文地址 http://tools.android.com/tech-docs/new-build-system/user-guide

GitBook 排版

在线阅读: http://gradle-guide.books.yourtion.com/

下载电子书: https://www.gitbook.com/book/yourtion/gradle-guide-book/details

- PDF: https://www.gitbook.com/download/pdf/book/yourtion/gradle-guide-book
- EPUB: https://www.gitbook.com/download/epub/book/yourtion/gradle-guide-book
- MOBI: https://www.gitbook.com/download/mobi/book/yourtion/gradle-guide-book

有修改建议优化，请直接**Fork：https://github.com/yourtion/GradleGuideBook** 进行修改并申请 **Pull Request。**

Yourtion

- yourtion@gmail.com
- https://github.com/yourtion

1 介绍

本文档适用于 Gradle plugin 0.9 版本，所以可能和我们1.0之前介绍的老版本有所不同。

1.1 新构建系统的目标

新构建系统的目标是：

- 可以很容易的重用代码和资源
- 可以很容易的创建应用的衍生版本，所以不管你是创建多个apk，还是不同功能的应用都很方便
- 可以很容易的配置、扩展以及自定义构建过程
- 和IDE无缝整合

1.2 Gradle是什么

Gradle 是一个非常优秀的构建系统工具，允许你通过插件的方式创建自定义的构建逻辑

Gradle 的以下特性让我们选择了它：

- 用过领域专用语言(DSL)描述和控制构建逻辑
- 构建文件基于 Groovy ，并且可以组合使用各种定义的元素，然后通过代码来控制这些DSL达到定制逻辑的目的
- 内建的基于 Maven 或者 Ivy 的依赖管理
- 使用非常灵活，Gradle 不会强制实现的方式，你可以使用最佳实践
- 插件能提供 DSL 以及 API 为构建文件使用
- 良好的工具 API 以供 IDE 集成

2 要求

- Gradle 1.10 或者 1.11 或者 1.12，并且使用 0.11.1 版本的插件
- SDK with Build Tools 要求 19.0.0，有些功能可能需要更新的版本

3 基础工程

一个Gradle工程是通过名字叫 `build.gradle` 的文件描述其构建过程的，该文字位于工程的根目录下。

3.1 基本的build文件

最基本的Java工程，其 `build.gradle` 非常简单：

```
apply plugin: 'java'
```

这里应用了Gradle提供的Java插件。该插件提供了构建和测试Java应用所需的一些东西。

一个最基本的Android工程的build.gradle如下：

```
buildscript {
    repositories {
        mavenCentral()
    }

    dependencies {
        classpath 'com.android.tools.build:gradle:0.11.1'
    }
}

apply plugin: 'android'

android {
    compileSdkVersion 19
    buildToolsVersion "19.0.0"
}
```

在 Android buid file 中，有3个主要组成部分。

`buildscript { ... }` 部分配置了驱动构建的代码。

在该部分中，定义配置使用了 Maven 中央仓库，并且声明依赖一个 Maven artifact(构件？)。 这个 artifact 是一个包含 0.11.1 版本的 Android Gradle 插件库。

注意： 这部分的配置只会影响构建过程的代码，和你的工程没有关系。工程会定义它自己的仓库和相关依赖。稍候会详细介绍。

接下来，android 插件被应用，这和上面的Java插件是一样的。

最后， `android { ... }` 这部分配置了 android 构建需要的所有参数。这里也是 Android DSL 的入口点。

默认情况下，只有编译的目标SDK、构建工具的版本是必需的。就是 `compileSdkVersion` 和 `buildtoolsVersion` 两个配置属性。

compilation target 和旧构建系统中的 `project.properties` 文件里 **target** 属性是一样的。这个新的属性和以前的 **target** 一样，可以指定一个 int 类型(api 级别)或者 string 类的值。

重要: 你应该只应用 android 插件就好了，不要同时应用 java 插件，因为这会导致构建错误

注意： 你还需要在同目录下添加一个 *local.properties* 文件，并通过 `sdk.dir` 属性配置所需的 SDK 的路径。除此之外，你也可以设置一个名为 `ANDROID_HOME` 环境变量。这两种方法都差不多，你可以选择自己喜欢的。

3.2 工程结构

上面说的 build 文件约定了一个默认的文件夹结构。Gradle 遵循约定优先于配置的原则，在可能的情况下提供合理的默认值。

基本的工程始于两个名为 "source sets" 的部分。也就是 main source code 和 test code。他们分别位于：

- src/main
- src/androidTest/

里面的每一个文件夹都对应相应的组件。

对于Java和Android这两个插件来说，他们的Java源代码和Java资源的位置是：

- java/
- resources/

对于 Android 插件来说，它还有以下特性文件和文件夹：

- AndroidManifest.xml
- res/
- assets/
- aidl/
- rs/
- jni/

注意： `src/androidTest/AndroidManifest.xml` 是不需要的，它会被自动创建。

3.2.1 配置结构

当默认的工程结构不适用的时候，你可能需要配置它。根据 Gradle 文档说明，可以通过如下方式重新配置Java工程的 `sourceSets`：

```
sourceSets {
    main {
        java {
            srcDir 'src/java'
        }
        resources {
            srcDir 'src/resources'
        }
    }
}
```

注意： `srcDir` 会添加指定的文件夹到现有的源文件夹列表中(Gradle 文档没有提到这个，但是的确是这样)。

要替换默认的源文件夹的话，可以给 `srcDirs` 指定一个路径数组 。下面使用对象调用另一种方式配置：

```
sourceSets {
    main.java.srcDirs = ['src/java']
    main.resources.srcDirs = ['src/resources']
}
```

想要了解更多的信息，请参考Gradle文档的Java插件部分。

Android插件也使用相似的语法，但是它有它自己的 *sourceSets* ,这些已经内置在 android 对象中了。

这儿有个示例，它使用了旧工程结构的源代码，并且重新映射了 androidTest *sourceSet* 到测试文件夹：

```
android {
    sourceSets {
        main {
            manifest.srcFile 'AndroidManifest.xml'
            java.srcDirs = ['src']
            resources.srcDirs = ['src']
            aidl.srcDirs = ['src']
            renderscript.srcDirs = ['src']
            res.srcDirs = ['res']
            assets.srcDirs = ['assets']
        }

        androidTest.setRoot('tests')
    }
}
```

注意： 因为旧结构中把所有的源文件(java, aidl, renderscript, and java resources)都放在同一文件夹下，所以我们需要重新映射这些 *sourceSet* 的新组件到同一src目录.

注意: setRoot()方法会移动整个 *sourceSet* (包括其下的子文件夹)到一个新文件夹。这里是移动src/androidTest/*到tests/*。

这些都是 Android 特有的，并不适用于 Java *sourceSets* 。

这是一个迁移的例子(译者注：比如从旧工程结构迁移过来)。

3.3 构建任务

3.3.1 通用任务

在构建文件中应用一个插件的时候会自动的创建一系列可运行的构建任务。Java plugin 和 the Android plugin 都可以做到这一点。以下是约定的一些任务：

- **assemble** 这个任务会汇集工程的所有输出。
- **check** 这个任务会执行所有校验检查
- **build** 这个任务会同时执行 **assemble** 和 **check** 任务
- **clean** 这个任务会清理工程的所有输出

事实上， **assemble** , **check** 以及 **build** 这三个任务并没有作任何事情，他们只是插件的引导任务，引导插件添加的其他任务去完成一些工作。

这样就可以允许你调用同样的任务，而不用管它是什么类型的工程或者应用了什么插件。

比如， 应用 *findbugs* 插件会创建一个任务，并且让 **check** 任务依赖它，这样当这个 **check** 任务被调用的时候，这个新创建的任务也会被调用.

在命令行中输入如下命令可以获取一些高级别的任务介绍。

```
gradle tasks
```

要查看所有的任务列表以及任务之间的依赖关系运行：

```
gradle tasks --all
```

注意： Gradle 会自动监控任务定义的输入和输出。

不做任何改变两次运行 **build** ， Gradle 会报告所有任务已经处于 UP-TO-DATE 状态，这意味着没有什么可做的。这使得任务之间可以正确的相互依赖，又不会导致其他不需要的操作执行。

3.3.2 Java 工程任务

Java plugin 创建了两个主要的任务，主要的引导任务都依赖他们。

- **assemble**
  - **jar** 这个任务创建所有输出
- **check**
  - **test** 这个任务运行所有测试

jar 任务直接或者间接的依赖其他任务：比如 **classes** 会编译所有Java代码.

testClasses 会编译所有测试，但是它很少使用，因为 **test** 这个任务依赖它(和 **classes** 差不多)。

通常情况下，你可能只用到 **assemble** 或者 **check** ,其他的任务不会使用。

你可以在这儿看到Java plugin的所有任务列表以及他们的依赖关系

3.3.3 Android 任务

Android plugin 使用了同样的约定规则以和其他插件保持兼容，并且又添加了一些额外的引导任务:

- **assemble** 这个任务会汇集工程的所有输出。
- **check** 这个任务会执行所有校验检查
- **connectedCheck** 运行 checks 需要一个连接的设备或者模拟器，这些checks将会同时运行在所有连接的设备上。
- **deviceCheck** 通过 API 连接远程设备运行 checks。它被用于 CI (译者注:持续集成)服务器上。
- **build** 这个任务会同时执行 **assemble** 和 **check** 任务
- **clean** 这个任务会清理工程的所有输出

这些新的引导任务是必须的，以便能够在没有连接的设备的情况下运行定期检查。

注意 **build** 既不依赖 **deviceCheck** ,也不依赖 **connectedCheck** 。

一个Android工程至少有两个输出:一个debug APK和一个 release APK。他们每一个都有自己的引导任务以便可以单独的构建他们：

- **assemble**
  - **assembleDebug**
  - **assembleRelease**

他们两个都依赖其他任务，这些任务执行很多必须的步骤以生成一个APK。 **assemble** 任务又依赖他们两个，所以执行 **assemble** 会生成两个 APK。

提示：在命令行下，Gradle 支持任务名称驼峰方式的快捷调用，比如：

```
gradle aR
```

和

```
gradle assembleRelease
```

是一样的，当然前提是没有其他任务匹配'aR'。

检验引导任务也有他们自己的依赖：

- **check**
  - **lint**
- **connectedCheck**
  - **connectedAndroidTest**
  - **connectedUiAutomatorTest** (尚未实现)
- **deviceCheck**
  - 这个依赖其他插件创建的时候实现的测试扩展点的哪些任务。

最后，插件会为所有的构建类型(**debug, release, test**)创建 install/uninstall 任务，也只有他们能被安装（需要签名）。

3.4 自定义构建

Android plugin 提供了大量的 DSL 能够让你直接基于构建系统定制很多事情。

3.4.1 Manifest选项

通过 DSL 可以配置 manifest 的如下选项：

- minSdkVersion
- targetSdkVersion
- versionCode
- versionName
- applicationId (更有效的 packageName -- 请看ApplicationId 与 PackageName获取更多信息)
- Package Name for the test application
- Instrumentation test runner

示例:

```
android {
    compileSdkVersion 19
    buildToolsVersion "19.0.0"

    defaultConfig {
        versionCode 12
        versionName "2.0"
        minSdkVersion 16
        targetSdkVersion 16
    }
}
```

android 元素里的 **defaultConfig** 负责定义所有的配置。

Android Plugin 以前的版本是使用 packageName 配置 manifest 的'packageName'属性。 从 0.11.0 开始，你应该在 `build.gradle` 里使用 applicationId 来配置 manifest 的'packageName'属性。 这是为了消除应用的 packageName（也就是 ID）和 java 的 packages 之间的歧义。

通过build文件定义的强大之处在于可以动态的被配置。 比如，可以从一个文件读取版本名字，或者使用一些其他的自定义的逻辑：

```
def computeVersionName() {
    ...
}

android {
    compileSdkVersion 19
    buildToolsVersion "19.0.0"

    defaultConfig {
        versionCode 12
        versionName computeVersionName()
        minSdkVersion 16
        targetSdkVersion 16
    }
}
```

注意： 不要使用在给定的范围内，和其他已经存在的 getters 方法冲突的方法名字。比如在 `defaultConfig { ...}` 中调用 `getVersionName()` 方法会自动的调用 `defaultConfig.getVersionName()` 方法，如果你也自定义一个这样的名字的方法，那么

你的方法不会调用。

如果一个属性没有通过DSL设置，则会使用它们的默认值。这里有个表格说明是如何处理的。

属性铭	DSL对象的默认值	默认值
versionCode	-1	如有有的话从 manifest 中读取
versionName	null	如有有的话从 manifest 中读取
minSdkVersion	-1	如有有的话从 manifest 中读取
targetSdkVersion	-1	如有有的话从 manifest 中读取
applicationId	null	如有有的话从 manifest 中读取
testApplicationId	null	applicationId + “.test”
testInstrumentationRunner	null	android.test.InstrumentationTestRunner
signingConfig	null	null
proguardFile	N/A (只能设置)	N/A (只能设置)
proguardFiles	N/A (只能设置)	N/A (只能设置)

如果你在构建脚本中使用自定义的逻辑获取这些属性的时候，那么第二列的值尤其重要。比如，你可能这样写：

```
if (android.defaultConfig.testInstrumentationRunner == null) {
    // assign a better default...
}
```

如果值一直为 null，那么在构建的时候，它将会被从第三列中获取的实际的默认值替换，但是在DSL元素中又不包含这个默认值，所以你无法查询到它。 这是为了防止解析应用的 manifest 文件，除非真的需要。

3.4.2 构建类型

默认情况下，Android plugin 会自动的设置工程，构建 release 和 debug 两个版本。 他们主要的差异主要在于是否可以在设备上调试应用以及APK如何签名。

debug 版本会被使用已知的名称/密码自动生成的密钥/证书签名。release 版本在构建过程中不会被签名，需要构建后再签名。

这些配置可以通过一个叫 **BuildType** 配置。默认情况下，已经创建了 **debug** 和 **release** 这两个实例。

Android plugin 允许自定义这两个示例，并且可以创建其他的 *Build Types* 。这些是可以在 **buildTypes** DSL容器中配置完成:

```
android {
    buildTypes {
        debug {
            applicationIdSuffix ".debug"
        }

        jnidebug.initWith(buildTypes.debug)
        jnidebug {
            packageNameSuffix ".jnidebug"
            jniDebuggable true
        }
    }
}
```

以上的片段实现了以下几点：

- 配置了默认的 **debug** 构建类型：
    - 设置包名为 `<app appliationId>.debug` 以便可以在同一设备上同时安装 *debug* 和 *release* 两个版本的APK
- 创建一个叫 **jnidebug** 新的 *Build Types* ， 并且配置它作为 **debug** 构建类型的一个副本
- 然后再配置 **jnidebug** ， 启用 JNI 组件的 debug 构建，并且添加一个不同的包名后缀

创建一个新的 *Build Types* 很简单，只需要在 **buildTypes** 容器下添加一个元素，然后调用 **initWith()** 或者使用一个闭包配置它。

这里有一些可能用到的属性以及他们的默认值：

属性名	**debug**时的默认值	**release**或者其他类型的默认值
debuggable	true	false
jniDebuggable	false	false
renderscriptDebuggable	false	false
renderscriptOptimLevel	3	3
applicationIdSuffix	null	null
versionNameSuffix	null	null
signingConfig	android.signingConfigs.debug	null
zipAlignEnabled	false	true
minifyEnabled	false	false
proguardFile	N/A (只能设置)	N/A (只能设置)
proguardFiles	N/A (只能设置)	N/A (只能设置)

除了这些属性外， 代码和资源也会影响到 *Build Types* 。 对于每一个 Build Type,都会创建一个新的匹配的 *sourceSet* ，默认位置是

```
src/<buildtypename>/
```

这意味着 *Build Type* 的名字不能是 *main* 和 *androidTest* (这两个已经被插件占用)，并且他们相互之间的名字必须唯一。

和其他的 source sets 一样，Build Type 的 source set 的位置可以被重定向：

```
android {
    sourceSets.jnidebug.setRoot('foo/jnidebug')
}
```

此外， 对于每一个 *Build Type* ， 都会新创建assemble\任务。

assembleDebug 和 **assembleRelease** 这两个任务已经讲过了， 这里讲的是他们是从哪来的。是在 **debug** 和 **release** 这两个 *Build Types* 被预先创建的时候。

提示：记得你可以通过输入 gradle aJ 来运行assembleJnidebug任务哦。

可能使用到的情况：

- 在 debug 模式下需要，但是在 release 下不需要的权限
- 自定义 debug 的实现
- 微 debug 默认使用不同的资源（比如一个资源的值是由签名的证书决定的）

BuildType 的代码/资源主要通过以下方式使用：

- manifest 被合并进 app 的 manifest
- 代码仅仅是作为一个额外的 source 文件夹(译者注：其实和自己新建一个 source 文件夹，然后在这个文件夹下新建包和类一样)
- 资源会覆盖 main 里的资源，替换现有的值

3.4.3 签名配置

要对一个应用签名，要求如下：

- 一个 keystore
- 一个 keystore 的密码
- 一个 key 的别名
- 一个 key 的密码
- 存储类型

位置、key 别名、key 密码以及存储类型一起组成了签名配置(*SigningConfig* 类型)

默认情况下， 已经有了一个 **debug** 的签名配置，它使用了 debug keystore，该 keystore 有一个已知的密码和默认的带有已知密码的 key。 debug keystore 位于$HOME/.android/debug.keystore，如果没有会被创建。

debug *Build Type* 被设置为自动使用 **debug** 签名配置。

你也可以创建其他的签名配置或者自定义内置的配置。可以通过 **signingConfigs** DSL容器实现。

```
android {
    signingConfigs {
        debug {
            storeFile file("debug.keystore")
        }

        myConfig {ss
            storeFile file("other.keystore")
            storePassword "android"
            keyAlias "androiddebugkey"
            keyPassword "android"
        }
    }

    buildTypes {
        foo {
            debuggable true
            jniDebuggable true
            signingConfig signingConfigs.myConfig
        }
    }
}
```

以上片段会把 debug keystore 的路径改为工程的根目录。这会自动的影响任何用到它的 *Build Types* ， 在这里影响到的是 **debug** *Build Type* 。

以前片段也创建了一个新的签名配置，并且被一个新的 *Build Type* 使用。

注意： 只有默认路径下的 debug keystores 才会被自动创建。如果改变了 debug keystore 的路径将不会在需要的时候创建。创建一个使用不同名字的 *SigningConfig* ， 但是用的是默认的 debug keystore 路径的话是会被自动创建的。也就是说，会不会被自动创建，和 keystore 的路径有关，和配置的名字无关。

说明： 通常情况下，会使用工程根目录的相对路径作为 keystores 的路径，但有时候也会用绝对路径，虽然这并不推荐(被自动创建的 debug keystore 除外)。

注意：如果已经你将这些文件放到版本控制中，你可能不想把密码存储在文件中。**Stack Overflow** 上有个帖子介绍可以从控制台或者环境变量中获取这些密码等信息。

http://stackoverflow.com/questions/18328730/how-to-create-a-release-signed-apk-file-using-gradle

我们以后更新这个指南的时候会添加更多的信息。

3.4.4 使用混淆

自从 Gradle plugin for ProGuard 4.10 版本以后，Gradle 开始支持混淆。如果通过 Build Type 的 *minifyEnabled* 属性配置了使用混淆后，The ProGuard plugin 会自动被应用，并且自动创建一些任务。

```
android {
    buildTypes {
        release {
            minifyEnabled true
            proguardFile getDefaultProguardFile('proguard-android.txt')
        }
    }

    productFlavors {
        flavor1 {
        }
        flavor2 {
            proguardFile 'some-other-rules.txt'
        }
    }
}
```

使用 buildTypes 以及 productFlavors 定义的规则文件可以轻松的生成多种版本。

有两个默认的规则文件

- proguard-android.txt
- proguard-android-optimize.txt

他们位于SDK中，使用 `getDefaultProguardFile()` 方法可以返回文件的全路经。除了是否启用优化之外，这两个文件的其他功能都是相同的。

3.4.5 清理资源

在构建的时候，你也可以自动的移除一些未使用的资源。更多信息，请参考资源清理文档

4 依赖，Android库工程以及多工程设置

Gradle 可以依赖其他的一些组件，这些组建可以是外部二进制包，也可以是其他 Gradle 工程。

4.1 依赖二进制包

4.1.1 本地包

要配置依赖一个外部库 jar 包，你可以在 **compile** 配置里添加一个依赖。

```
dependencies {
    compile files('libs/foo.jar')
}

android {
    ...
}
```

注： **dependencies** DSL 元素是标准 Gradle API 的一部分，并不属于 **android** 的元素。

compile 配置用来编译 main application，它里面的一切都会被添加到编译的 classpath 中，并且也会被打包到最终的 APK 中。

这里还有添加依赖时其他的配置：

- **compile**： main application
- **androidTestCompile**： test application
- **debugCompile**： debug Build Type
- **releaseCompile**： release Build Type

因为要构建生成一个 APK，必然会有相关联的 *Build Type*， APK默认配置了两个(或者更多)编译配置：compile和\Compile。创建一个新的 *Build Type* 的时候会自动创建一个基于它名字的编译配置。

当一个 debug 版本需要一个自定义库(比如报告崩溃)，但是 release 版本不需要或者需要一个不同版本的库的时候，会显得非常有用。

4.1.2 远程 artifacts

Gradle 支持从 Maven 和 Ivy 仓库获取 artifacts。

首先必须把库添加到列表中，并且以 Maven 或者 Ivy 的定义方式定义需要的依赖。

```
repositories {
    mavenCentral()
}


dependencies {
    compile 'com.google.guava:guava:11.0.2'
}

android {
    ...
}
```

注： **mavenCentral()** 是指定仓库URL的快捷方式。Gradle 同时支持远程和本地两种仓库 注：Gradle 遵循依赖的传递性。这就意味着如果依赖本身会依赖其他东西，这些也会被拉取过来。

更多关于 dependencies 的设置信息，请参见Gradle 用户指南，以及DSL文档

4.2 多工程设置

Gradle 工程可以通过多工程配置依赖其他的Gradle工程

多工程配置通常把所有的工程作为根目录的子文件夹。

比如，下面的工程结构：

```
MyProject/
  app/
  libraries/
     lib1/
     lib2/
```

我们可以识别这三个工程。Gradle 会通过如下名字引用他们：

```
:app
:libraries:lib1
:libraries:lib2
```

每个工程都有属于它自己的 `build.gradle` 文件定义如何构建它自己。

此外，在工程根目录下有个叫 *settings.gradle* 的文件会定义所有工程。

文件结构如下：

```
MyProject/
 settings.gradle
 app/
    build.gradle
 libraries/
    lib1/
       build.gradle
    lib2/
       build.gradlef
```

settings.gradle 里的内容非常简单：

```
include ':app', ':libraries:lib1', ':libraries:lib2'
```

这里定义了哪个文件是一个Gradle工程。

:app 工程也可能会依赖一些库工程，可以通过如下脚本声明依赖：

dependencies { compile project(':libraries:lib1') }

更多关于多工程的配置请参考这里

4.3 库工程

在上面的多工程配置中， **:libraries:lib1** 和 **:libraries:lib2** 可能是Java工程，并且 **:app** Android工程会用到他们生成的jar报。

但是，如果你想共享访问 Android API 的代码或者使用 Android 的样式资源，那么这个库工程就不能是通常的 Java 工程，而应该是 Android 库工程。

4.3.1 创建一个库工程

一个 Android 库工程和一个 Android 工程非常相似，只有一点差别。

因为构建一个库和构建一个应用不太一样，所以它使用了一个不同的插件。这两个插件(构建应用和库的)大部分代码都是一样，并且都是通过 `com.android.tools.build.gradle` jar 包提供。

```
buildscript {
    repositories {
        mavenCentral()
    }

    dependencies {
        classpath 'com.android.tools.build:gradle:0.5.6'
    }
}

apply plugin: 'android-library'

android {
    compileSdkVersion 15
}
```

这里创建一个基于 API 15 级别编译的库工程。*SourceSets*， 以及依赖的处理方式和应用工程是一样并且可以以同样的方式进行自定义。

4.3.2 普通工程和库工程的不同

库工程的 main 输出是一个 .aar 报(这个一个标准的 Android 存档).它由编译后的代码（比如 jar 文件或者 .so 文件）以及资源文件(manifest, res, assets)组成。

库工程也可以生成一个测试apk，可以独立于应用进行测试。

它有相同的引导任务(**assembleDebug** , **assembleRelease**)，所以他和一般的工程没有什么不同。

其余的，基本上都和应用一样了。他们都有 build types 和 product flavors，可以生成多个版本的 aar。

注意 *Build Type* 大部分配置并不适用于库工程。不过你可以依据库工程是被其他工程依赖还是测试，然后通过自定义库工程 *sourceSet* 改变它的内容。

4.3.3 引用一个库工程

引用一个库工程和引用其他工程是一样的：

```
dependencies {
    compile project(':libraries:lib1')
    compile project(':libraries:lib2')
}
```

说明：如果你有多个依赖库工程， 顺序是很重要的。这和旧构建系统中在 `project.properties` 文件中定义的依赖顺序是一样的。

4.3.4 库工程发布

默认情况下库工程只能发布 *release* 版本。这个版本用于所有工程的引用，和工程本身要构建什么样的版本无关。这是属于 Gradle 的限制，我们正在努力消除这个限制。

你可以通过如下方式控制发布的各种版本

```
android {
    defaultPublishConfig "debug"
}
```

注意这里的发布配置的名字使用的是一个完整的版本名字。 *Release* 和 *debug* 仅仅适用于没有其他版本的时候使用。如果你想用其他版本代替默认发布的版本，你可以这么做：

```
android {
    defaultPublishConfig "flavor1Debug"
}
```

发布库工程的所有不同版本也是可能的。我们计划在在一般的工程对工程依赖中(就像上面说的)允许这么做。但是这还不被Gradle允许(我么正在努力修复这个问题)。

默认情况下，没有启用发布所有不同的版本功能，你可以这么启用他们：

```
android {
    publishNonDefault true
}
```

要意识到，发布多个不同的版本意味着会有多个 aar 文件，而不是一个 aar 文件中包含多个不同的版本。每个aar包只包含一个版本。 发布一个版本意味着要生成一个可用的 aar 文件作为 Gradle 工程的输出构件。它既可以被发不到一个 maven 仓库，也可以被其他工程依赖引用。

Gradle有默认'构件'的概念，当作如下配置时会用到：

```
compile project(':libraries:lib2')
```

要创建另外一个已发布构建的依赖，你需要指定需要哪一个：

```
dependencies {
    flavor1Compile project(path: ':lib1', configuration: 'flavor1Release')
    flavor2Compile project(path: ':lib1', configuration: 'flavor2Release')
}
```

重要： 注意发布配置的是一个完整的版本，包括 build type，并且需要像上面一样被引用。

重要： 当启用了非默认发布的时候，Maven 的发布插件将会发布其他版本作为扩展包（按分级）。这意味着和 maven 上的版本不一定兼容。你应该发布一个单独的版本到仓库中或者为工程间的依赖启用所有的发布配置。

5 测试

构建的测试应用已经被集成在应用工程里，不需要再创建一个单独的测试工程。

5.1 基础介绍和配置

正如上面讲到的， **main** *sourceSet* 的旁边就是 **androidTest** *sourceSet*， 默认的路径是 src/androidTest/ 从这个 *sourceSet* 可以构建一个能安装到设备上的测试apk， 该 apk 使用 Android 测试框架测试应用。这里包括单元测试、集成测试以及后续的UI自动化测试。 这个测试 *sourceSet* 不必包含 `AndroidManifest.xml` 文件，因为它会自动生成。

测试应用有如下值可以配置：

- **testPackageName**
- **testInstrumentationRunner**
- **testHandleProfiling**
- **testFunctionalTest**

如前所见，这些可以在 **defaultConfig** 对象里配置。

```
android {
    defaultConfig {
        testPackageName "com.test.foo"
        testInstrumentationRunner "android.test.InstrumentationTestRunner"
        testHandleProfiling true
        testFunctionalTest true
    }
}
```

在测试应用的 manifest 文件中，instrumentation 节点的 targetPackage 属性值会自动的被测试应用的包名填充，即使是通过 defaultConfig 或者 Build Type 对象定义的。这也是 manifest 文件自动生成的一个原因。

此外，*sourceSet* 也可以配置它自己的依赖。 默认情况下， 应用以及它自己的依赖会被添加测试应用的 classpath 中，但是也可以进行扩展

```
dependencies {
    androidTestCompile 'com.google.guava:guava:11.0.2'
}
```

测试应用通过 **assembleTest** 任务来构建。它并不依赖 main 的 **assemble** 任务，并且不能自动调用，需要手动运行。

当前只能同时测试一个 *Build Type*， 默认是 **debug** *Build Type*， 但是也可以被重新配置

```
android {
    ...
    testBuildType "staging"
}
```

5.2 运行测试

正如前面所提到的，引导任务 **connectedCheck** 需要一个已经连接的设备才能运行。 这会依赖 **androidTest**， 所以 **androidTest** 也会被运行。这个任务做了以下事情：

- 确保应用和测试应用已经被构建(依赖 **assembleDebug** 和 **assembleTest**)
- 安装这两个应用
- 运行测试
- 卸着这两个应用

如果同时有多个连接的设备，那么所有的测试会在所有的设备上运行。不管在哪个设备上，只要有一个测试失败，那么整个构建就是失败的。

所有的测试结果已XML的格式被存储在 build/androidTest-results 目录下。(这类似于 jUnit ,它的结果存储在 build/test-results 目录下)

当然，你可以自己设置

```
android {
    ...

    testOptions {
        resultsDir = "$project.buildDir/foo/results"
    }
}
```

android.testOptions.resultsDir 的值是通过 **Project.file(String)** 得到。

5.3 测试Android库

测试 Android 库工程的方式和应用工程是一样。

仅有的不同就是整个库(包括它的依赖)会作为一个依赖库被自动的添加到测试应用中。测试APK的测试结果不仅包括它自己代码的测试，还包括 Android 库的以及库的所有依赖的测试。 库的 manifest 被合并到测试应用的 manifest 中(这种情况就和任何工程引用这个库是一样的)

androidTest 任务的职责也不一样了，它仅仅负责安装(和卸载)测试 APK (因为也没有其他APK需要安装)

其他的都和测试应用差不多。

5.4 测试报告

当运行单元测试的时候，Gradle 会生成一份 HTML 报告以便于查看测试结果。 Android plugins 在这个基础上扩展了 HTML 报告，以合并所有已连接设备上的测试结果。

5.4.1 单工程报告

在运行测试的时候工程会自动的生成报告，默认位置是：

```
build/reports/androidTests
```

这和 jUnit 报告的位置 build/reports/tests 很相似，其他报告的位置通常是 build/reports/\/

报告的位置也可以自定义

```
android {
    ...

    testOptions {
        reportDir = "$project.buildDir/foo/report"
    }
}
```

报告会合并运行在不同设备上的测试结果。

5.4.2 多工程报告

在一个既有应用工程又有库工程的多工程里，当在同时运行所有测试的时候，生成一个包含所有测试结果的报告是非常有用的。

为了达到这一目的，需要同一构件中的另外一个插件，可以通过如下方式应用：

```
buildscript {
    repositories {
        mavenCentral()
    }

    dependencies {
        classpath 'com.android.tools.build:gradle:0.5.6'
    }
}

apply plugin: 'android-reporting'
```

这应该被应用到根目录下,也就是和 `settings.gradle` 相邻的 `build.gradle` 文件中.

然后,在根目录打开终端,输入如下命令运行所有测试并且收集报告:

```
gradle deviceCheck mergeAndroidReports --continue
```

注: `--continue` 选项确保所有测试都被执行,即使测试是失败的.否则的话第一个失败的测试会中断运行,那么就可能会有一些工程的测试不会被运行.

5.5 Lint支持

从0.7.0版本之后,你可以为一个特定的变种版本运行 lint ,也可以为所有变种版本都运行.在这种情况下,它会产生一个报告指出给定的变种版本的问题.

你可以像下面一样通过 lintOptions 自定义 lint .一般情况下,你只需要配置其中的一部分.以下是展示所有可用的 lint 配置项.

```
android {
    lintOptions {
        // set to true to turn off analysis progress reporting by lint
        quiet true
        // if true, stop the gradle build if errors are found
        abortOnError false
        // if true, only report errors
        ignoreWarnings true
        // if true, emit full/absolute paths to files with errors (true by default)
        //absolutePaths true
        // if true, check all issues, including those that are off by default
        checkAllWarnings true
        // if true, treat all warnings as errors
        warningsAsErrors true
        // turn off checking the given issue id's
        disable 'TypographyFractions','TypographyQuotes'
        // turn on the given issue id's
        enable 'RtlHardcoded','RtlCompat', 'RtlEnabled'
        // check *only* the given issue id's
        check 'NewApi', 'InlinedApi'
        // if true, don't include source code lines in the error output
        noLines true
        // if true, show all locations for an error, do not truncate lists, etc.
        showAll true
        // Fallback lint configuration (default severities, etc.)
        lintConfig file("default-lint.xml")
        // if true, generate a text report of issues (false by default)
        textReport true
        // location to write the output; can be a file or 'stdout'
        textOutput 'stdout'
        // if true, generate an XML report for use by for example Jenkins
        xmlReport false
        // file to write report to (if not specified, defaults to lint-results.xml)
        xmlOutput file("lint-report.xml")
        // if true, generate an HTML report (with issue explanations, sourcecode, etc)
        htmlReport true
        // optional path to report (default will be lint-results.html in the builddir)
        htmlOutput file("lint-report.html")

   // set to true to have all release builds run lint on issues with severity=fatal
   // and abort the build (controlled by abortOnError above) if fatal issues are found
   checkReleaseBuilds true
        // Set the severity of the given issues to fatal (which means they will be
        // checked during release builds (even if the lint target is not included)
        fatal 'NewApi', 'InlineApi'
        // Set the severity of the given issues to error
        error 'Wakelock', 'TextViewEdits'
        // Set the severity of the given issues to warning
        warning 'ResourceAsColor'
        // Set the severity of the given issues to ignore (same as disabling the check)
        ignore 'TypographyQuotes'
    }
}
```

6 构建变种版本

新构建系统的目标之一就是为同一个应用创建不同的版本。

主要有两个使用场景：

1. 同一个应用的不同版本。比如一个免费的版本和一个付费的专业版本。
2. 同一个应用被打包成多个不同的 apk 以发布到 Google Play 商店。详情请见 http://developer.android.com/google/play/publishing/multiple-apks.html
3. 综合第1条和第2条。

我们的目标就是基于同一个工程生成不同的APK，而不是使用一个单独的库工程和两个以上的应用工程组合生成 APK 的方式。

6.1 产品定制

一个 product flavor 定义了可以通过工程构建应用的自定义版本。一个独立的工程可以定义不同的flavor改变生成的应用。

这种被设计的新概念对于版本间差异非常小的时候很有用。如果“这是同一个应用吗？”的答案是肯定的话，那么这种方式的确比使用库工程的方式要好得多。(译者注：以前的方法要生成多个包，可能是从采用多个不同的应用工程+一个库工程的方式，现在这种新的方式比我们以前的老方式好多了)

Product flavors 是通过 **productFlavors** DSL容器定义的：

```
android {
    ....

    productFlavors {
        flavor1 {
            ...
        }

        flavor2 {
            ...
        }
    }
}
```

这里创建了两个 flavors，分别是 flavor1 和 flavor2 . 注意：flavors 的名字不能和已存在的 *Build Type* 名字或者 **androidTest** *sourceSet* 冲突。

6.2 构建类型+产品定制=构建变种版本

正如我们前面看到的，每一个 *Build Type* 都会生成一个新的APK。

Product Flavors 也是这么做的：工程的输出将会尽可能的组合 *Build Types* 和 *Product Flavors* 的输出。

每一种组合(Build Type, Product Flavor)就是 构建变种

比如，以默认的 **debug** 和 **release** *Build Types* 为例，上面的例子会生成四个 *Build Variants* ：

- Flavor1 - debug
- Flavor1 - release
- Flavor2 - debug
- Flavor2 - release

没有 flavors 的工程仍然是有 *Build Variants* 的，只是使用的是默认的 flavor 和配置，并且没有名字，所以 variants 的列表看起来和 Build Types 列表一样。

6.3 产品定制配置

每一个 flavors 都可以通过一个闭包配置：

```
android {
    ...

    defaultConfig {
        minSdkVersion 8
        versionCode 10
    }

    productFlavors {
        flavor1 {
            packageName "com.example.flavor1"
            versionCode 20
        }

        flavor2 {
            packageName "com.example.flavor2"
            minSdkVersion 14
        }
    }
}
```

要知道的是 **android.productFlavors.*** 是 *ProductFlavor* 类型的，和 **android.defaultConfig** 对象具有相同的类型，者意味着他们有相同的属性。

defaultConfig 为所有的 flavor 提供了基本的配置，每一个 flavor 也都可以重新设置覆盖这些默认值。在上面的例子中，最终的配置如下：

- flavor1
  - packageName: com.example.flavor1
  - minSdkVersion: 8
  - versionCode: 20
- flavor2
  - packageName: com.example.flavor2
  - minSdkVersion: 14
  - versionCode: 10

通常情况下，*Build Type* 配置会覆盖其他配置，比如，*Build Type* 的 **packageNameSuffix** 会追加到 *Product Flavor* 的 **packageName** 之后。

也有一些情况是在 *Build Type* 和 *Product Flavor* 中都可以设置，在这种情况下，视情况而定。

比如， **signingConfig** 就是这么一个属性。 通过设置 **android.buildTypes.release.signingConfig** ， 可以为所有的 release 包共享相同的 SigningConfig ， 也可以单独通过设置 android.productFlavors..*signingConfig* 为每一个 *release* 包指定他们自己的 SigningConfig* 。

6.4 Sourcesets和依赖

类似 *Build Types* ， *Product Flavors* 也可以通过他们自己的 *sourceSets* 控制代码和资源。

上面的例子会创建4个 *sourceSets* ：

- **android.sourceSets.flavor1** ， 位置是src/flavor1/
- **android.sourceSets.flavor2** ， 位置是src/flavor2/
- **android.sourceSets.androidTestFlavor1** ， 位置是src/androidTestFlavor1/
- **android.sourceSets.androidTestFlavor2** ， 位置是src/androidTestFlavor2/

这些 *sourceSets* 和 **android.sourceSets.main** 以及 *Build Type sourceSet* 一起构建APK。

当处理所有的 *sourcesets* 以构建一个单独的APK的时候，下面的规则会被应用：

- 所有的源代码(src/*/java)会以多文件夹的方式一起被使用生成一个输出。
- 所有Manifest文件会合并成一个 manifest 文件。这允许 *Product Flavors* 有一些不同的组件定义或者权限声明， 类似于 *Build Types* 。
- 所有的资源(Android res 和 assets)都会遵循优先级覆盖的原则， *Build Type* 会覆盖 *Product Flavorg* ， 最后又都会覆盖 **main** *sourceSet* .
- 每一个 *Build Variant* 会基于资源生成他们自己的R类(或者生成其他的源代码)，variant之间不会共享。

最后，像 *Build Types, Product Flavors* 也可以有他们自己的依赖。比如，如果 flavor 用来生成一个广告 ap 和一个付费的 app，其中一个 flavor 可能需要依赖一个广告 SDK，另外一个则不需要。

```
dependencies {
    flavor1Compile "..."
}
```

在这种特定的情况下， `src/flavor1/AndroidManifest.xml` 文件可能需要包含访问网络的权限声明。

此外，也会为每一个 variant 创建一个 sourcesets：

- **android.sourceSets.flavor1Debug** ,位置是src/flavor1Debug/
- **android.sourceSets.flavor1Release** ,位置是src/flavor1Release/
- **android.sourceSets.flavor2Debug** ,位置是src/flavor2Debug/
- **android.sourceSets.flavor2Release** ,位置是src/flavor2Release/

这些 sourcesets 拥有比 build type 更高的优先级，并且允许在 variant 级别上做一些定制。

6.5 构建和任务

我们在前面说过，每一个 *Build Type* 都会创建它自己的 assemble\ 任务,但是 Build Variants 的任务则是 Build Type 和 Product Flavor 的组合。

当 Product Flavors 被使用的时候，更多的 assemble-type 任务被创建，他们是：

1. assemble\ 允许直接构建一个 variant 版本。例如 **assembleFlavor1Debug**
2. assemble\ 允许根据给定的 Build Type 构建所有的APK。例如 **assembleDebug** 会同时构建 Flavor1Debug 和 Flavor2Debug 两个 variant 版本。
3. assemble\ 允许根据给定的 flavor 构建所有的APK。例如 **assembleFlavor1** 会同时构建 Flavor1Debug 和 Flavor1Release 两个 variant 版本。

assemble 任务会尽可能的构建所有 variant 版本。

6.6 测试

测试多 flavor 工程和测试普通的工程差不多。

androidTest sourceset 对所有的 flavor 来说是通用的测试，而每个 flavor 也可以有他们自己的测试。

正如前面所提到的，每一个 flavor 都可以创建自己的测试 *sourceSets* ：

- **android.sourceSets.androidTestFlavor1**， 位置是src/androidTestFlavor1/
- **android.sourceSets.androidTestFlavor2**， 位置是src/androidTestFlavor2/

同样的，他们也可以有他们自己的依赖：

```
dependencies {
    androidTestFlavor1Compile "..."
}
```

运行测试可以通过主的 **deviceCheck** 引导任务，当使用 flavor 的时候，也可以通过主的 **androidTest** 引导任务来执行。

每一个flavor都有他们自己运行测试的任务：androidTest\。例如：

- **androidTestFlavor1Debug**
- **androidTestFlavor2Debug**

同样的，每一个 variant 也都有测试APK任务、安装以及卸载任务：

- **assembleFlavor1Test**
- **installFlavor1Debug**
- **installFlavor1Test**
- **uninstallFlavor1Debug**
- ...

最终的HTML报告会根据 flavor 合并生成。

测试结果以及报告的位置如下，第一个是每个 flavor 的结果，然后是合并起来的：

- build/androidTest-results/flavors/\
- build/androidTest-results/all/
- build/reports/androidTests/flavors\
- build/reports/androidTests/all/

自定义路径的话，也只是改变根目录，仍然会创建每个 flavor 的子文件夹并且合并测试结果以及报告。

6.7 多种定制的版本

有些情况下，人们想基于不同的标准创建同一应用的几个不同的版本。 例如，Google Play 里的 multi-apk 支持4种不同的过滤器。为每一个过滤器创建不同的 APK 就需要用到多维度的 *Product Flavor*了。

考虑到一个游戏有一个演示版本和一个付费版本，并且在 multi-apk 支持中需要用到 ABI 过滤器。3个 ABI 和两个版本的情况下，就会有6个 APK 生成(没有计算不同的 Build Type的variant 版本)。 然而，对于三个 ABI 来说，他们的付费版本的代码都是一样的，因此只是简单的创建6个 flavor 并不是一个好办法。 相反的，使用两个 flavor 维度，并且自动构建所有可能的 variants。

这个功能通过 Flavor Dimensions 能实现。Flavors 会被指定到特定的维度。

```
android {
    ...

    flavorDimensions "abi", "version"

    productFlavors {
        freeapp {
            flavorDimension "version"
            ...
        }

        x86 {
            flavorDimension "abi"
            ...
        }
    }
}
```

android.flavorDimensions 数据定义了可能用到的唯独以及顺序。每一个定义的 *Product Flavor* 都会被指定一个纬度。

从 Product Flavors [freeapp, paidapp]、[x86, arm, mips]、[debug, release] Build Types 维度，会有以下 build variant 被创建：

- x86-freeapp-debug
- x86-freeapp-release
- arm-freeapp-debug
- arm-freeapp-release
- mips-freeapp-debug
- mips-freeapp-release
- x86-paidapp-debug
- x86-paidapp-release
- arm-paidapp-debug
- arm-paidapp-release
- mips-paidapp-debug
- mips-paidapp-release

通过 **android.flavorDimensions** 定义的维度的顺序是非常重要的。

每一个 variant 都会被如下几个 *Product Flavor* 对象配置：

- **android.defaultConfig**
- abi 维度
- version 维度

维度的顺序决定了哪一个 flavor 会覆盖哪一个 flavor，这对于资源来说非常重要，因为 flavor 会替换掉定义在低优先级 flavor 中的值。

flavor 维度首先使用高优先级的定义。在这里是：

```
abi > version > defaultConfig
```

Multi-flavors 工程也有额外的 sourcesets。类似 variant 的 sourcesets，只是没有 build type。

- **android.sourceSets.x86Freeapp**， 位置是src/x86Freeapp/
- **android.sourceSets.armPaidapp**， 位置是src/armPaidapp/
- 等等...

这允许你在 flavor-combination 级别上进行定制。他们比普通的 flavor sourcesets 优先级高，但是比 build type sourcesets 优先级低。

7 高级构建定制

7.1 构建选项

7.1.1 Java 编译选项

```
android {
    compileOptions {
        sourceCompatibility = "1.6"
        targetCompatibility = "1.6"
    }
}
```

默认值是1.6。这个设置会影响所有编译 java 源代码的任务。

7.1.2 aapt选项

```
android {
    aaptOptions {
        noCompress 'foo', 'bar'
        ignoreAssetsPattern "!.svn:!.git:!.ds_store:!*.scc:.*:<dir>_*:!CVS:!thumbs.db:!picasa.ini:!*~"
    }
}
```

这会影响所有使用 appt 的任务。

7.1.3 dex选项

```
android {
    dexOptions {
        incremental false
        preDexLibraries = false
        jumboMode = false
        javaMaxHeapSize "2048M"
    }
}
```

这会影响所有使用 dex 的任务

7.2 操纵任务

普通的 Java 工程有一个有限的任务集合，这些任务相互配合创建一个输出。 **classes** 是一个编译Java源代码的任务。 在 `build.gradle` 中通过脚本访问和使用 classes 任务是很简单的。可以通过 `project.tasks.classes` 快捷访问。

对于 Android 工程来说就比较复杂了，因为可能有很多相同的任务，他们的名字是基于 *Build Types*和*Product Flavors* 生成的。

为了解决这个问题， **android** 对象提供了两个属性：

- applicationVariants (仅仅适用于 app plugin)
- libraryVariants (仅仅适用于 library plugin)
- testVariants (两种 plugin 都适用)

ApplicationVariant, LibraryVariant, and TestVariant 这三个对象都会分别返回一个DomainObjectCollection。

请注意访问这些集合中的任何一个都会触发生成所有的创建。这意味着访问这些集合后无须重新配置就会产生。

DomainObjectCollection 允许直接的访问所有对象，或者通过更为方便的过滤器访问。

```
android.applicationVariants.each { variant ->
    ....
}
```

这三个variant类都具有以下属性：

属性名	属性类型	说明
name	String	variant的名字，必须是唯一的。
description	String	variant的可读性的描述
dirName	String	variant的子文件夹名称，必须是惟一的。可能还不止一个，比如 “debug/flavor1”
baseName	String	variant输出的的基本名称，必须是唯一的。
outputFile	File	variant的输出，是一个可读写的属性
processManifest	ProcessManifest	处理manifest的任务
aidlCompile	AidlCompile	编译AIDL文件的任务
renderscriptCompile	RenderscriptCompile	编译Renderscript文件的任务
mergeResources	MergeResources	合并资源的任务
mergeAssets	MergeAssets	合并资源的任务
processResources	ProcessAndroidResources	处理和编译资源的任务
generateBuildConfig	GenerateBuildConfig	生成BuildConfig类的任务
javaCompile	JavaCompile	编译Java代码的任务
processJavaResources	Copy	处理Java资源的任务
assemble	DefaultTask	这个variant的assemble引导任务

ApplicationVariant类还有以下属性：

属性名	属性类型	说明
buildType	BuildType	variant的BuildType。
productFlavors	List\	variant的ProductFlavors，不能为空，但可以是空值
mergedFlavor	ProductFlavor	android.defaultConfig和variant.productFlavors合并
signingConfig	SigningConfig	这个variant使用的SigningConfig对象
isSigningReady	boolean	如果为true则说明variant已经具备签名所需的一切信息
testVariant	BuildVariant	将会测试这个variant的TestVariant
dex	Dex	dex代码的任务，如果variant是一个库可以为null
packageApplication	PackageApplication	生成最终AP看的任务，如果variant是一个库可以为null
zipAlign	ZipAlign	zip压缩apk的任务，如果variant是一个库或者APK不能被签名可以为null
install	DefaultTask	安装任务，可以为null。
uninstall	DefaultTask	卸载任务

LibraryVariant类还有以下属性：

属性名	属性类型	说明
buildType	BuildType	variant的BuildType
mergedFlavor	ProductFlavor	就是defaultConfig
testVariant	BuildVariant	将要测试这个variant的Build Variant
packageLibrary	Zip	把库打包成一个AAR存档的任务，如果不是一个库值为Null

TestVariant类还有以下属性：

属性名	属性类型	说明
buildType	BuildType	variant的BuildType。
productFlavors	List\	variant的ProductFlavors，不能为空，但可以是空值
mergedFlavor	ProductFlavor	android.defaultConfig和variant.productFlavors合并
signingConfig	SigningConfig	这个variant使用的SigningConfig对象
isSigningReady	boolean	如果为true则说明variant已经具备签名所需的一切信息
testVariant	BaseVariant	将会测试这个variant的BaseVariant
dex	Dex	dex代码的任务，如果variant是一个库可以为null
packageApplication	PackageApplication	生成最终AP看的任务，如果variant是一个库可以为null
zipAlign	ZipAlign	zip压缩apk的任务，如果variant是一个库或者APK不能被签名可以为null
install	DefaultTask	安装任务，可以为null。
uninstall	DefaultTask	卸载任务
connectedAndroidTest	DefaultTask	在已连接的设备上运行android测试的任务
providerAndroidTest	DefaultTask	使用扩展的API运行android测试的任务

Android特定任务类型的API

- ProcessManifest
  - File manifestOutputFile
- AidlCompile
  - File sourceOutputDir
- RenderscriptCompile
  - File sourceOutputDir
  - File resOutputDir
- MergeResources
  - File outputDir
- MergeAssets
  - File outputDir
- ProcessAndroidResources
  - File manifestFile
  - File resDir
  - File assetsDir
  - File sourceOutputDir
  - File textSymbolOutputDir
  - File packageOutputFile
  - File proguardOutputFile
- GenerateBuildConfig
  - File sourceOutputDir
- Dex
  - File outputFolder
- PackageApplication
  - File resourceFile
  - File dexFile
  - File javaResourceDir
  - File jniDir
  - File outputFile
    - To change the final output file use "outputFile" on the variant object directly.
- ZipAlign
  - File inputFile
  - File outputFile
    - To change the final output file use "outputFile" on the variant object directly.

每一个任务类型的 API 都会因为 Gradle 的工作方式以及 Android plugin 的设置受到限制。 首先，Gradle 限制只能配置任务的输入/输出以及一些可选的标志。所以，这里的这些任务只能定义输入/输出。

其次，大多数任务的输入并不唯一，通常会混合 sourceSets，Build Types 以及 Product Flavors。为了保持构建简单以及可读性，我们的目标是让开发者通过 DSL 通过这些对象修改构建，而不是深入的了解任务的输入和选项进而修改它们。

还要注意的是，除了 ZipAlign 任务类型，其他所有的任务都需要设置私有数据让他们运行。这就意味着不能基于这些类型手动的创建新的任务。

对于 Gradle 的任务（DefaultTask, JavaCompile, Copy, Zip），请参考 Gradle 文档。

7.3 BuildType and Product Flavor 的属性参考

即将推出。

对于 Gradle 的任务（DefaultTask, JavaCompile, Copy, Zip），请参考 Gradle 文档。

7.4 使用sourceCompatibility 1.7

基于 Android KitKat (buildToolsVersion 19)开发的时候，你能用 diamond operator, multi-catch, strings in switches, try with resources 等等这些新的特性。要做到这些，你需要把下面的配置添加到你的构建文件中：

```
android {
    compileSdkVersion 19
    buildToolsVersion "19.0.0"

    defaultConfig {
        minSdkVersion 7
        targetSdkVersion 19
    }

    compileOptions {
        sourceCompatibility JavaVersion.VERSION_1_7
        targetCompatibility JavaVersion.VERSION_1_7
    }
}
```

需要注意的是你也可以把 **minSdkVersion** 设置为19之前的版本，这样的话你只能使用除 try with resources 之外的语言特性。如果你想使用 try with resources，你需要设置 **minSdkVersion** 为19。

你还需要确认 Gradle 使用的 JDK1.7 或者之后的版本。（并且 Android Gradle plugin 同样也需要0.6.1或者之后的版本）